La programación orientada a objetos

La programación estructurada es muy sencilla e intuitiva. En ella simplemente tenemos un problema y lo subdividimos en problemas cada vez más pequeños. Usamos funciones para colocar zonas especializadas de código o código que se usa constantemente en su interior.

El paradigma estructurado es correcto, pero tiene algunas limitaciones y algunos problemas cuando el programa a realizar es grande. Unos de estos problemas se conoce como la corrupción de información. Supongamos que tenemos una variable y ésta es utilizada en diversas partes del programa. Una forma sencilla para poder tener acceso a la variable es declararla de forma global, así todo el programa la conoce. Al hacer esto cualquier parte del programa no sólo puede leerla sino también modificarla. Debido a este esquema, una parte del programa puede cambiarse sin que otra sepa de este cambio y así se producen errores en la información. Para evitar esto es posible pasarle la variable como parámetro a quien la necesite, pero esto complica su administración.

Los programas estructurados muy grandes también son difíciles de extender y mantener, y llevar a cabo un cambio puede ser costoso. Generalmente, un cambio en una parte del programa produce cambios en otras partes que quizá no estén relacionadas directamente. En general, los programas estructurados son poco flexibles.

La programación estructurada es buena para los programas pequeños y para aprender las bases de programación. Sin embargo, el mundo actual pide el desarrollo orientado a objetos. Como recomendación, después de este libro, podemos aprender análisis y diseño orientado a objetos y luego continuar aprendiendo más C# con todas las técnicas orientadas a objetos. Esto ya sería un nivel avanzando - experto de programación.

En la programación orientada a objetos tomamos una idea diferente en la resolución de los problemas. En lugar de subdividirlos, lo que hacemos es ver cuáles son los componentes u objetos que componen el problema y la forma cómo interactúan. Nosotros programaremos estos objetos y haremos que se comuniquen entre sí. Cuando los objetos hagan su labor y se comuniquen, el problema estará resuelto.

Las clases

El componente principal de la programación orientada a objetos es la **clase**. Nosotros podemos pensar en la clase como si fuera un plano por medio del cual podemos crear objetos, por ejemplo, pensemos en una casa. Para hacer una casa lo primero que hacemos es pensar en los cuartos que deseamos que tenga y luego diseñamos un plano. El plano no es la casa, ya que no podemos vivir ni actuar en él. Sin embargo, éste nos proporciona las características de la casa. Por medio del plano podemos construir la casa y en esta construcción sí podemos llevar a cabo nuestras actividades.

El plano sería equivalente a la clase y la casa construida al **objeto**. La clase es un plano, una descripción, y el objeto tiene esas características y puede llevar a cabo trabajo concreto. Si necesitásemos hacer otra casa igual, no sería necesario hacer un nuevo plano, simplemente tomaríamos el plano ya realizado y crearíamos otra casa. Una clase nos puede servir para crear muchos objetos independientes pero que tienen las mismas características. El proceso de crear un objeto a partir de una clase es lo que conocemos como **instanciación.**

Adentro de la clase, nosotros colocaremos información y más importante aún, los **métodos** que trabajan sobre esta información, es decir, que los **datos** y los métodos que los procesan están contenidos dentro de una sola unidad. A esto lo llamamos **encapsulamiento**. Al encapsular datos y métodos los protegemos contra la corrupción de información.

Los objetos se comunicarán por medio del uso de **mensajes**. En estos mensajes es posible solicitarle un dato a otro objeto, pedirle que lleve a cabo un proceso, etcétera.

Los datos

Los datos son la información con la que trabajará la clase. La clase solamente debe tener los datos que necesita para poder llevar a cabo su trabajo. Declarar un dato es muy similar a declarar una variable, pero al igual que en las estructuras, necesitamos indicar el acceso ya que tenemos básicamente tres tipos de acceso: **público**, **privado** y **protegido**. Cuando nosotros tenemos un dato con acceso público cualquier elemento del exterior, como la función **`Main()`** o algún otro objeto, puede acceder al dato, leerlo y modificarlo. Cuando tenemos el acceso privado solamente los métodos definidos dentro de la clase podrán leerlo o modificarlo. El acceso protegido es un poco más avanzado y está por afuera de los límites de este libro.

Un punto muy importante con relación a los datos que no debemos olvidar es que los datos definidos en la clase son conocidos por todos los métodos de la misma clase. Es decir, actúan como si fueran globales para la clase. Cualquier método puede acceder a ellos directamente sin necesidad de que los pasemos como parámetro.

En algunos casos podremos colocar el acceso a nuestros datos como público, aunque preferentemente no lo haremos. Si nos excedemos o usamos el acceso público en un mal diseño, corremos el riesgo de corrupción de información. Por lo general,

Es una característica de la programación orientada a objetos. Ésta permite crear una nueva clase que hereda las características (**datos y métodos**) de otra clase, de forma tal que solamente tengamos que agregar los elementos necesarios para la nueva. Es una gran forma de reutilización de código si se usa en forma adecuada.

nuestros datos serán privados, aunque esto puede parecer un problema ya que si el exterior necesita alguna información calculada por el objeto no podrá tener acceso a ella. Para resolver esto hacemos uso de las funciones de interfaz.
Una **función de interfaz** es aquella que puede ser invocada desde el exterior y que regresa una copia del valor de algún dato dentro del objeto. También podemos usarla para colocar un valor determinado en un dato. La ventaja que nos da la función de interfaz es que podemos administrar el acceso a nuestra información, y podemos colocar dentro de ésta código de seguridad que verifique o valide la información que entra o sale. De esta forma evitamos la corrupción de información.

Los métodos

Los métodos son las funciones que llevan a cabo el proceso o la lógica de la clase, y crear un método dentro de la clase es muy parecido a la forma que hemos utilizado anteriormente. Los métodos también tendrán un tipo de acceso, al igual que los datos. Trabajarán sobre los datos de la clase. No hay que olvidar que todos los métodos conocen todos los datos definidos dentro de la clase, y pueden recibir parámetros y regresar valores. A un dato definido dentro de la clase no necesitamos pasarlo como parámetro ya que el método lo conoce. Solamente los métodos que necesiten ser invocados desde el exterior deben tener acceso público. Si el método sólo se invoca desde el mismo interior de la clase su acceso debe ser privado. Esto lo hacemos con fines de seguridad y para mantener el encapsulamiento correctamente.

Cómo declarar la clase y los datos

La declaración de la clase es un proceso sencillo. Las clases se declaran dentro de un **namespace** y cualquiera que tenga acceso a ese namespace puede crear objetos de la clase. No olvidemos que la clase es como el plano y los objetos son realmente los que usamos para llevar a cabo el trabajo.

Para declarar la clase tendremos un esquema como el siguiente:

```
class nombre
{

     // datos
…
…
     // métodos
…
…
```

```
}
```

El **nombre** de la clase puede ser cualquier nombre válido dentro de C#. El nombre debe ser único para el namespace, es decir no podemos tener dos clases que se llamen igual adentro del mismo namespace. La clase necesita un **bloque de código** y en su interior llevamos a cabo las declaraciones de los elementos que la componen. Generalmente, declaramos primero los datos. Esto nos permite tener un programa organizado y luego facilita la lectura. Además, es posible declarar los métodos implementados antes.
La mejor forma de hacer esto es por medio de un ejemplo. Supongamos que crearemos un programa que calcula el área y el volumen de cubos y prismas rectangulares. Como en esta ocasión lo hacemos vía programación orientada a objetos, lo primero que hacemos es pensar en los objetos que componen el problema.
Los objetos son: **cubo** y **prisma rectangular**. Ahora que ya conocemos los objetos, debemos pensar en los datos y las operaciones que se realizan sobre éstos.
Para el cubo necesitaremos como datos primordiales la longitud de su lado, el área y su volumen. En el caso del prisma requerimos aún más datos, que son el ancho, el alto, el espesor, el área y el volumen.
Enseguida debemos pensar en las operaciones que se realizan sobre estos datos. El cubo es más sencillo de resolver, ya que solamente necesitamos dos métodos, uno llamado **CalculaArea()** y el otro llamado **CalculaVolumen()**. Para comprender mejor las clases y por motivos ilustrativos el prisma necesitará tres métodos. El primero se llamará **CalculaArea()**, el segundo **CalculaVolumen()** y el tercero **AreaRectangulo()**. Como el área del prisma es igual a la suma de los rectángulos de sus caras nos apoyaremos en este método para calcularla.

Podemos diagramar las clases si hacemos uso de **UML**. La clase del cubo quedará como se muestra en la siguiente figura:

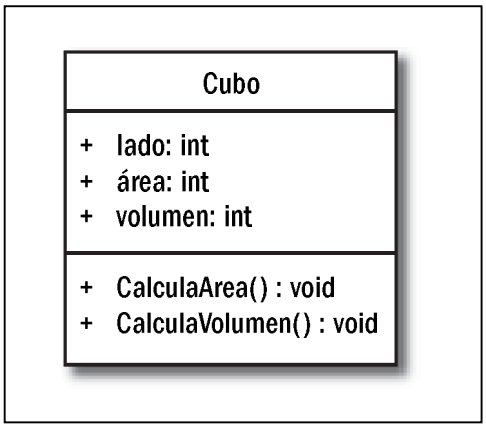

Figura 1. *Éste es el diagrama de clases para el cubo. Podemos ver todos los elementos que lo conforman.*

De igual forma es posible crear el diagrama de clases correspondiente al prisma:

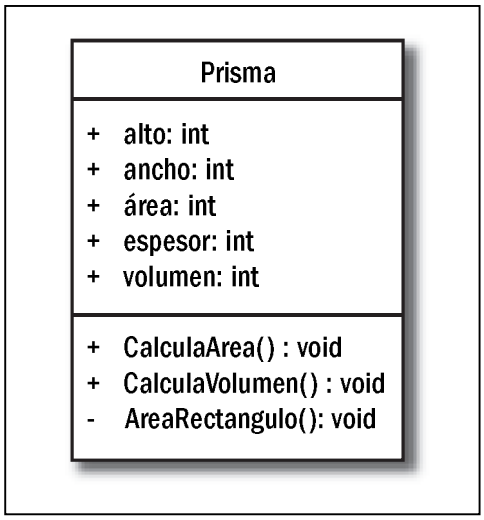

Figura 2. *Esta figura nos muestra el diagrama de clases para el prisma.*

Los diagramas anteriores se pueden leer fácilmente. Hay un rectángulo que representa a la clase. El rectángulo está dividido en tres secciones. La sección superior, que es utilizada para colocar el nombre de la clase, la sección intermedia, que se usa para indicar los datos que tendrá la clase y la sección inferior, que es para indicar cuáles son los métodos a utilizar.
El acceso se muestra por medio de los signos **+** o **-**. Cuando usamos el signo más estamos indicando que ese dato o método tiene un acceso del tipo público. En el caso de que coloquemos el signo menos, el acceso es del tipo privado.
El formato de los datos lleva primero el acceso seguido del nombre del dato. Luego se coloca **:** y el tipo que tiene este dato. El tipo puede ser nativo o definido por el programador. En el caso de los métodos, primero indicamos el acceso seguido del nombre del método. Como ninguno de nuestros métodos necesita parámetros, entonces los dejamos vacíos. El tipo del método, es decir el tipo de valor que regresa, se indica por medio de **:** seguido del tipo. En nuestro ejemplo hasta el momento ningún método regresa algo, por lo que todos son de tipo **void.**
Ahora podemos comenzar a declarar nuestras clases:

UML es un lenguaje unificado de modelado, un lenguaje visual que nos sirve para llevar a cabo diagramas y modelado de sistemas. Resulta muy útil en la programación y el diseño orientado a objetos, ya que facilita el diseño y la depuración de la aplicación aun antes de que se escriba una línea de código. Es recomendable buscar información adicional sobre éste y aprenderlo.

```
class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;
}

class prisma
{
    // Declaramos los datos
    public int ancho;
    public int alto;
    public int espesor;
    public int area;
    public int volumen;
}
```

Como podemos observar, declaramos dos clases. En cada una hemos declarado los datos que le corresponden. La declaración de los datos es muy similar a la declaración de variables, y si prestamos atención, podemos notar que ambas clases tienen nombres que se repiten en sus datos como **area** y **volumen**. Esto es posible porque cada clase es una entidad diferente, y estos nombres nunca se confundirán entre sí. Continuemos con nuestra clase cubo. Ahora lo que necesitamos hacer es colocar los métodos de la clase. Las operaciones son muy sencillas. Nuestra clase cubo quedará como se muestra en el siguiente bloque de código:

```
class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;

    // Método para calcular el área
    public void CalculaArea()
    {
        area = (lado * lado) * 6;
    }
```

```
        // Método para calcular el volumen
        public void CalculaVolumen()
        {
            volumen = lado * lado * lado;
        }
    }
```

La declaración de los métodos se hace adentro del bloque de código de la clase y cada uno de ellos tiene su propio bloque de código. Como podemos observar, usan los datos directamente. Esto se debe a que cualquier dato declarado en la clase es conocido por todos los métodos de esa clase. Más adelante continuaremos con la clase **prisma.**

Cómo crear la instancia de nuestra clase

Ya tenemos la clase cubo que es el plano donde se indica qué datos tiene y qué información utilizará con ellos. Ahora tenemos que construir el objeto, que es quien llevará a cabo el trabajo en realidad. Para instanciar un objeto de la clase cubo debemos utilizar el código que se muestra a continuación:

```
cubo miCubo = new cubo();
```

Quien realiza realmente la instanciación es **new** y esa instancia queda referenciada como **miCubo**. A partir de **miCubo** podremos empezar a trabajar con él.

Cómo asignarles valores a datos publicos

Como ya tenemos la instancia, ya podemos comenzar a trabajar con ella. Lo primero que haremos será asignarle un valor al dato **lado**. Haremos la asignación por medio del operador de asignación **=** (igual), pero también deberemos indicar a cuál de todos los datos de la clase vamos a acceder. Esto lo hacemos con el

El acceso privado es utilizado cuando hacemos uso de la herencia. Si declaramos un dato como protegido, la propia clase y las clases que hereden de ellas podrán acceder a él, leerlo y modificarlo. Todas las demás clases lo verán como si su acceso fuera privado y no podrán acceder a él directamente.

operador punto. Por ejemplo, asignemos el valor de **5** al **lado**, realizando esto como vemos en el código a continuación:

```
miCubo.lado = 5;
```

Cómo invocar métodos de un objeto

Cuando invocamos el método de un objeto, éste ejecuta el código que tiene en su interior. Desde el exterior de la clase solamente podemos invocar métodos que sean públicos. La invocación del método es muy similar a lo que aprendimos en el **Capítulo 3**, con todos los casos que vimos. En este caso sólo tenemos que indicar con qué objetos trabajaremos, seguido el operador **.** y el nombre del método con sus parámetros, si los necesita.
Invoquemos los métodos para calcular el área y el volumen del cubo:

```
            // Invocamos los métodos
            miCubo.CalculaArea();
            miCubo.CalculaVolumen();
```

Cómo imprimir un dato público

Como ya tenemos los valores calculados, ahora los podemos mostrar. Para esto los usaremos como cualquier variable normal, pero debemos indicar el objeto con el que trabajamos, seguido del operador **.** y en nombre del dato.

```
            // Desplegamos los datos
            Console.WriteLine("Area={0}, Volumen={1}", miCubo.area,
               miCubo.volumen);
```

Nuestro programa queda de la siguiente manera:

```
class cubo
    {
        // Declaramos los datos
        public int lado;
        public int area;
        public int volumen;
```

```
        // Método para calcular el área
        public void CalculaArea()
        {
            area = (lado * lado) * 6;
        }

        // Método para calcular el volumen
        public void CalculaVolumen()
        {
            volumen = lado * lado * lado;
        }
    }

    class prisma
    {
        // Declaramos los datos
        public int ancho;
        public int alto;
        public int espesor;
        public int area;
        public int volumen;
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Instanciamos a la clase cubo
            cubo miCubo = new cubo();

            // Asignamos el valor del lado
            miCubo.lado = 5;

            // Invocamos los métodos
            miCubo.CalculaArea();
            miCubo.CalculaVolumen();

            // Desplegamos los datos
            Console.WriteLine("Area={0}, Volumen={1}", miCubo.area,
              miCubo.volumen);
```

```
        }
    }
```

Si ejecutamos el programa obtendremos lo siguiente:

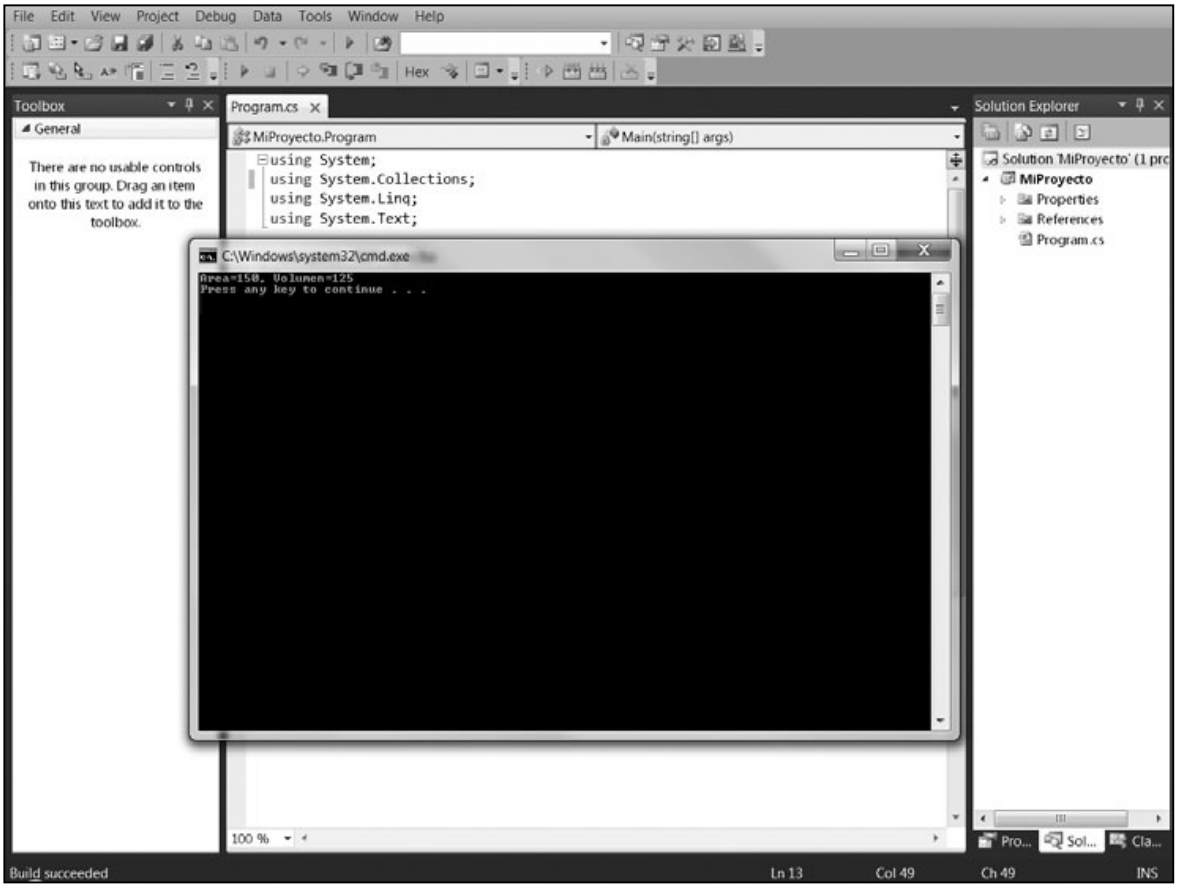

Figura 3. *Vemos el resultado de la ejecución de nuestro programa.*

Una de las ventajas que tiene la programación orientada a objetos es la reutilización de código. Si necesitáramos dos cubos o más, simplemente creamos nuevas instancias. Cada una de ellas tendría en su interior sus propias variables y podría llevar a cabo los cálculos que fueran necesarios.
Por ejemplo, modifiquemos el programa para que se tengan dos cubos. El segundo cubo estará en la instancia **tuCubo** y tendrá un valor de **lado** de **8.**

```
        static void Main(string[] args)
        {
            // Instanciamos a la clase cubo
            cubo miCubo = new cubo();
            cubo tuCubo = new cubo();
```

```
            // Asignamos el valor del lado
            miCubo.lado = 5;
            tuCubo.lado = 8;

            // Invocamos los métodos
            miCubo.CalculaArea();
            miCubo.CalculaVolumen();
            tuCubo.CalculaArea();
            tuCubo.CalculaVolumen();

            // Desplegamos los datos
            Console.WriteLine("Mi cubo Area={0}, Volumen={1}",
               miCubo.area, miCubo.volumen);
            Console.WriteLine("Tu cubo Area={0}, Volumen={1}",
               tuCubo.area, tuCubo.volumen);
        }
```

Este cambio solamente fue necesario en **Main()**, ya que todo el comportamiento que necesitamos se encuentra en la clase. Ejecutemos el programa y veamos el resultado.

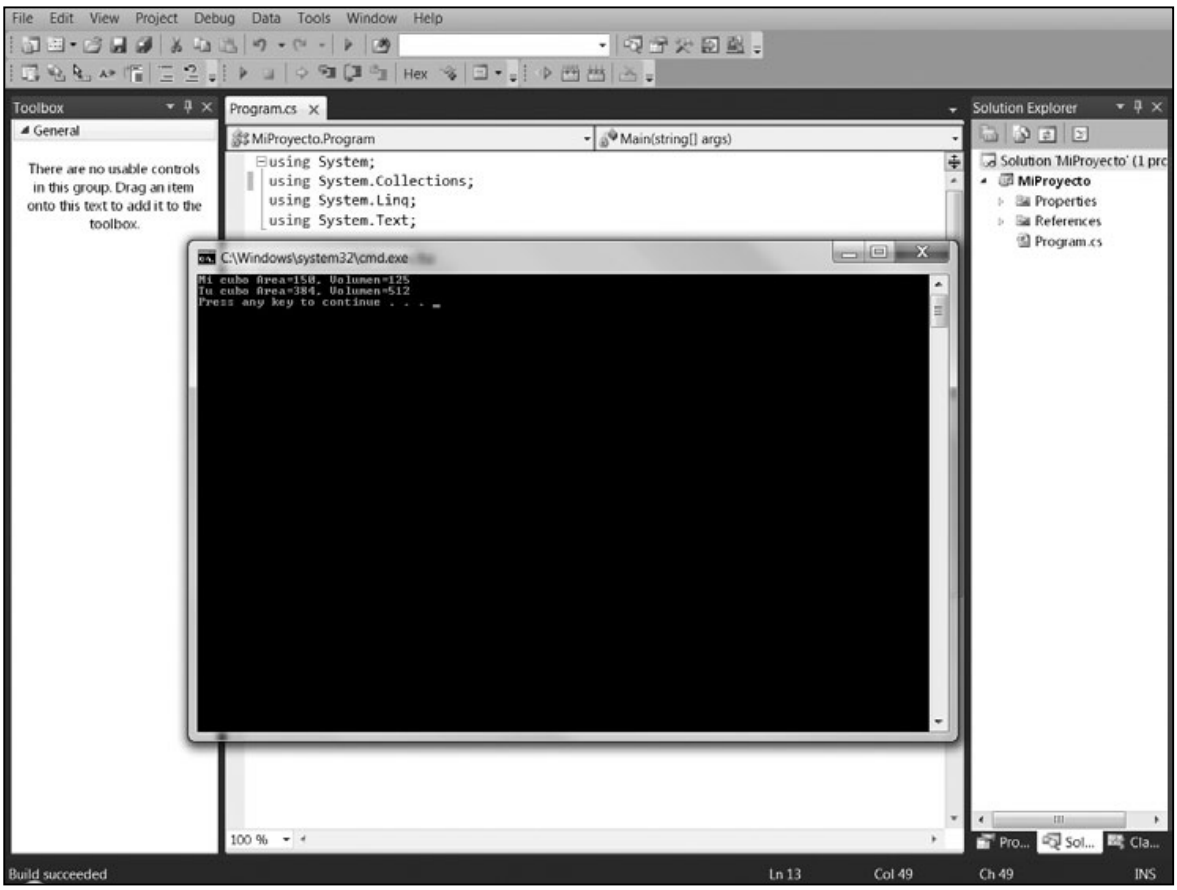

Figura 4. *Aquí observamos los resultados calculados por ambas instancias.*

Protección de datos y creación de propiedades

Con la clase cubo hemos visto cómo crear una clase sencilla que ya tiene funcionalidad. Sin embargo, presenta un problema. El primero es que todos sus datos son públicos, lo que nos puede llevar a corrupción de información. Para proteger los datos tenemos que hacerlos privados y proveer una función de interfaz a aquellos a los que se necesita acceder por el exterior.

Para tener una comparación, trabajaremos sobre nuestra clase prisma. Lo primero que hacemos es cambiar el acceso de los datos a privado.

```
class prisma
{
    // Declaramos los datos
    private int ancho;
    private int alto;
    private int espesor;
    private int area;
    private int volumen;
}
```

Las propiedades son funciones de interfaz. Nos permiten acceder a los datos privados de una manera segura y controlada, pero van más allá de simples funciones ya que también nos brindan una forma de acceso intuitiva y sencilla.

La propiedad puede ser de varios tipos: **lectura**, **escritura** y la combinación de ambas(**lectura-escritura**). Una propiedad de lectura solamente nos permite leer el dato, pero no podemos agregarle información. Una propiedad que es de tipo escritura sólo nos permite colocar información en el dato, pero no podemos leerlo. La propiedad de **lectura-escritura** permite llevar a cabo ambas acciones.

Para lograr esto, la propiedad tendrá dos métodos. El método relacionado con la lectura se conoce como **get** y el relacionado con la escritura es **set**. Dependiendo de cuál método coloquemos en la propiedad, será su tipo.

La propiedad de éste tiene la siguiente forma de declaración:

```
public tipo nombre {

    get
    {
    …
    …
    return x;
```

```
        }

        set
        {
        …
        …
        x=value;
        }

}
```

Las propiedades son públicas para poder llamarlas desde el exterior de la clase. El tipo está referenciado al tipo del valor que leerá o colocará, ya sea entero, flotante, doble, etcétera. En su interior tenemos **get**, donde colocamos el código para sacar un valor de la clase por medio de **return**, y a **set**, donde ponemos el código necesario para introducir un valor en la clase.

Empecemos por crear propiedades para la clase prisma. Lo primero que tenemos que preguntarnos es a qué datos se necesita acceder por el exterior y qué tipo de acceso requieren. Podemos ver que los datos **ancho**, **alto** y **espesor** necesitarán de escritura, pero también de lectura. Esto es en caso de que necesitemos saber las dimensiones. A sus propiedades las llamaremos: **Ancho**, **Alto** y **Espesor.**

Los otros datos que necesitan tener una propiedad son **area** y **volumen**, pero en este caso solamente necesitamos leerlos. No tiene sentido escribir sobre esos datos, ya que la clase calculará sus propios valores.

Ahora que ya sabemos cuáles son las propiedades necesarias podemos decidir si es necesario algún tipo de validación. Sabemos que no podemos tener prismas con cualquiera de sus lados con valor de **0** o negativos y ése es un buen punto para validar. Si el usuario diera un valor incorrecto, entonces colocaremos por **default** el valor **1**. Hay que recordar que esto lo hacemos como ejemplo y cada aplicación puede tener sus propias reglas de validación para la información.

Cuando usemos el método set tendremos una variable previamente definida por el lenguaje que se llama **value**. Esta variable representa el valor que el usuario asigna y podemos usarlo en la lógica que necesitemos.

Nuestras propiedades de lectura y escritura quedan de la siguiente forma:

```
// Definimos las propiedades
        public int Ancho
        {
```

```
        get
        {
            return ancho;
        }

        set
        {
            if (value <= 0)
                ancho = 1;
            else
                ancho = value;

        }

    }

    public int Alto
    {
        get
        {
            return alto;
        }

        set
        {
            if (value <= 0)
                alto = 1;
            else
                alto = value;

        }
```

En ocasiones, dentro de los métodos **get** y **set** de la propiedad, tendremos un **return** o una asignación. Aunque esto es correcto, no debemos olvidar que en las propiedades podemos colocar cualquier lógica válida de C# que nos permita validar la información que sale o entra. Esto hará que nuestro código sea más seguro y evitaremos problemas con nuestra información.

```
        }

        public int Espesor
        {
            get
            {
                return espesor;
            }

            set
            {
                if (value <= 0)
                    espesor = 1;
                else
                    espesor = value;

            }

        }
```

Ahora crearemos las propiedades de sólo lectura para el área y el volumen:

```
        public int Area
        {
            get
            {
                return area;
            }
        }

        public int Volumen
        {
            get
            {
                return volumen;
            }
        }
```

Como estas propiedades son de sólo lectura únicamente llevan el método **get.**

Cómo acceder a las propiedades

Acceder a las propiedades es muy sencillo ya que únicamente necesitamos colocar el objeto con el que queremos trabajar seguido del operador **.** y el nombre de la propiedad. La asignación se lleva a cabo por medio del operador **=.**
Por ejemplo, si deseamos indicar que el ancho tiene **5** unidades, hacemos lo siguiente:

```
miPrisma.Ancho=5;
```

Y si deseamos imprimir el valor de la propiedad **Volumen**:

```
Console.WriteLine("El volumen es {0}",Volumen);
```

Métodos públicos y privados

Como mencionamos anteriormente, los métodos pueden ser públicos o privados. Los primeros pueden ser invocados desde el exterior del objeto y los privados solamente desde su interior. Al programar la única diferencia es el tipo de acceso que colocamos. En nuestro ejemplo necesitamos dos métodos públicos para invocar al cálculo del área y el volumen, y un método privado que nos apoyará en el cálculo del área. Veamos cómo programarlos:

```
        public void CalculaVolumen()
        {
            volumen = ancho * alto * espesor;
        }

        public void CalculaArea()
        {
            int a1 = 0, a2 = 0, a3 = 0;

            a1 = 2 * CalculaRectangulo(ancho, alto);
            a2 = 2 * CalculaRectangulo(ancho, espesor);
            a3 = 2 * CalculaRectangulo(alto, espesor);

            area = a1 + a2 + a3;
        }

        private int CalculaRectangulo(int a, int b)
```

```
        {
            return (a * b);
        }
```

Los métodos son muy sencillos, lo importante es notar que el método **CalculaRectangulo()** tiene acceso privado, por lo que nadie del exterior puede invocarlo. Sin embargo, **CalculaArea()** lo invoca sin problemas ya que pertenece a la misma clase.

Convertir un objeto a cadena

Tenemos varias opciones para imprimir la información que guarda un objeto en la consola. La primera consiste en leer la propiedad e imprimir de la forma tradicional. Otra opción puede ser crear un método dentro del objeto que se especialice en la impresión de la información. Esto puede ser útil si deseamos imprimir tan solo uno o algunos datos.

La última opción y la que aprenderemos a usar es la de programar el método **ToString()** para la clase. Esto ya lo hemos hecho en el capítulo anterior para las estructuras. El mecanismo es similar, simplemente tenemos que implementar una versión del método **ToString()** para nuestra clase. Este método regresa una cadena que contiene la información en el formato que deseamos y tampoco necesita recibir ningún parámetro.

Este método se implementa adentro de la clase y cada clase en la aplicación lo puede tener de ser necesario. Entonces, cuando necesitemos imprimir los contenidos del objeto simplemente lo invocaremos e imprimiremos la cadena resultante. El método debe tener acceso público ya que es necesario que el exterior pueda invocarlo.

En nuestro caso, el método quedaría de la siguiente manera:

```
        public override string ToString()
        {
            String mensaje = "";
            mensaje += "Ancho " + ancho.ToString() + " Alto " +
              alto.ToString() + " Espesor " + espesor.ToString();
            mensaje += " Area " + area.ToString() + " Volumen " +
              volumen.ToString();
            return mensaje;
        }
```

En este ejemplo hacemos uso de la concatenación para poder generar la cadena que el método regresará. Hacemos esto para comparar con el uso de **StringBuilder**, que se

ha utilizado en el capítulo donde hablamos de las estructuras. La concatenación suele utilizarse para agrupar registros obtenidos de una base de datos. Podemos hacer uso de cualquiera de estos métodos según necesitemos de ellos. La impresión de los contenidos del objeto apoyándonos en este método puede ser de la siguiente forma:

```
Console.WriteLine(miPrisma.ToString());
```

Si lo deseamos, podemos probar a éste en nuestro programa, y ver que obtendremos el resultado de la siguiente figura:

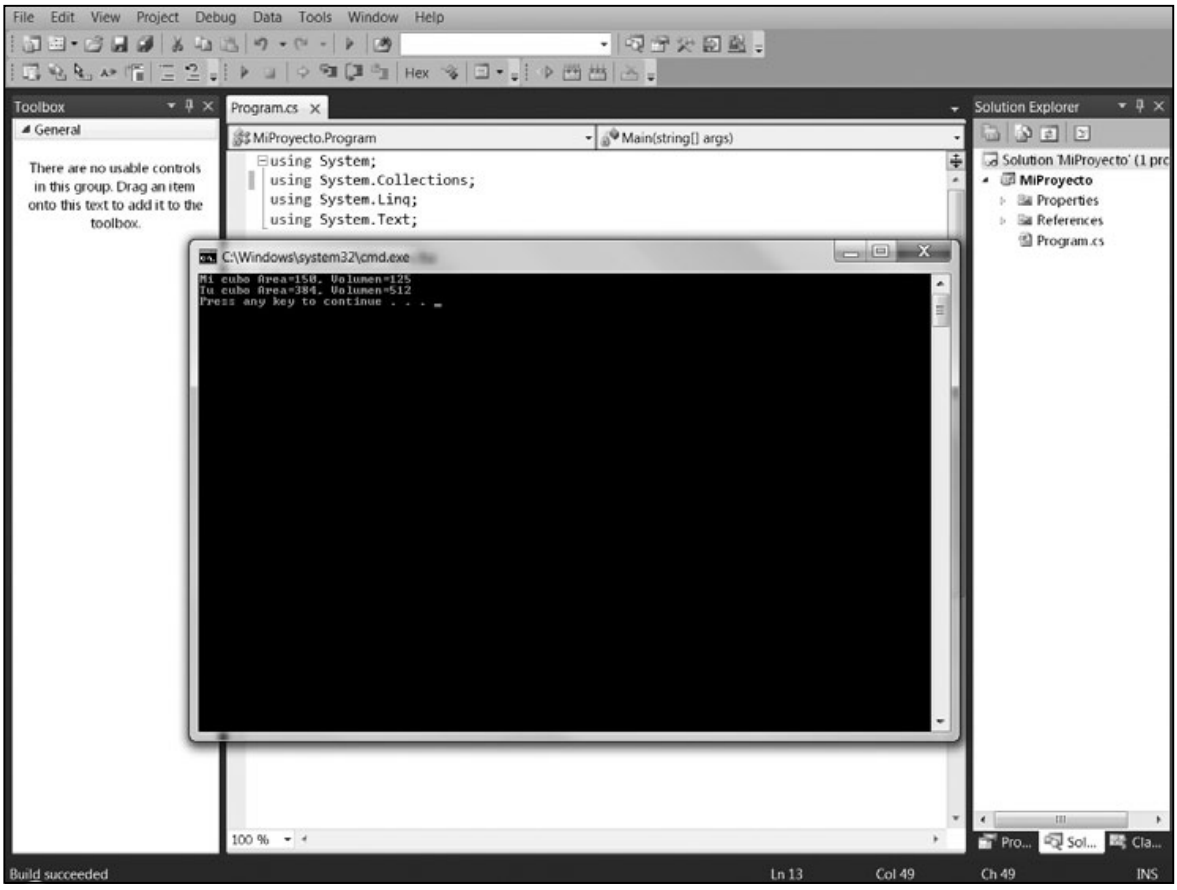

Figura 5. *Podemos observar cómo hemos impreso los contenidos del objeto apoyándonos en el método* ***ToString().***

Para imprimir solamente algunos datos

Si solamente necesitamos imprimir algunos datos, entonces es necesario crear un método especializado para ello. Este método deberá tener acceso público para que pueda ser invocado desde el exterior, y en el caso de llegar a necesitarlo, puede prepararse el método para recibir parámetros, aunque esto no es necesario. Supongamos que deseamos tener disponible un método que sólo imprima los resultados para el área y el volumen.

El trozo de código para ello, es el que mostramos el el siguiente bloque:

```
public void ImprimeResultado()
{
   Console.WriteLine("El área es {0}, el volumen es {1}", area,
       volumen);
}
```

Como observamos, el código del método es muy sencillo. Igualmente podemos probarlo en nuestra aplicación y obtener el siguiente resultado en la consola.

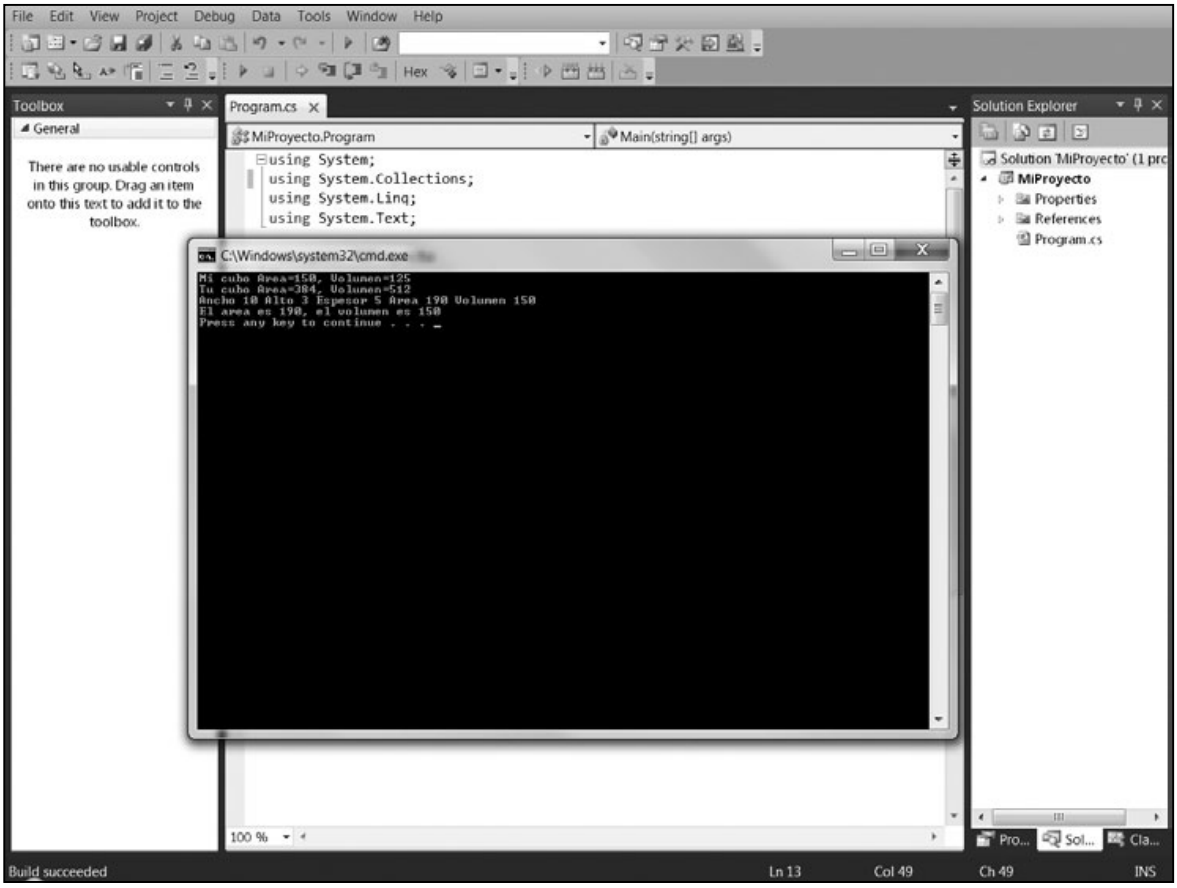

Figura 6. *Ahora podemos comparar la impresión con la ayuda de* ***ToString()*** *y del método especializado.*

El constructor en las clases

El constructor es un método especial que podemos utilizar con las clases. Éste generalmente es usado para inicializar los valores de los datos con los que trabajará el objeto. La forma como lo utilizamos con las clases es equivalente a la forma como lo utilizamos con las estructuras en el capítulo anterior. El constructor es un método especial y tiene ciertas características que lo distinguen de los demás métodos.

Su primera característica es que tiene el mismo nombre de la clase y su segunda característica más importante es que no tiene **tipo**, es decir, que no solamente no regresa nada, sino que no tiene tipo alguno.
El constructor es invocado en forma automática cuando el objeto es instanciado, ya que esto nos da la oportunidad de llevar a cabo cosas en el instante que se instancia el objeto, como por ejemplo, hacer inicializaciones. El constructor puede tener en su interior cualquier código válido de C# y también puede tener parámetros o no. Si utilizamos los parámetros tendremos que pasar los valores necesarios en el momento en el que instanciamos el objeto.
Veamos un ejemplo de constructor para nuestra clase prisma. En este constructor no utilizaremos parámetros, veamos el siguiente ejemplo:

```
public prisma()
{
     // Datos necesarios
     String valor="";

     // Pedimos los datos
     Console.WriteLine("Dame el ancho");
     valor=Console.ReadLine();
     ancho=Convert.ToInt32(valor);

     Console.WriteLine("Dame el alto");
     valor=Console.ReadLine();
     alto=Convert.ToInt32(valor);

     Console.WriteLine("Dame el espesor");
     valor=Console.ReadLine();
     espesor=Convert.ToInt32(valor);

}
```

Mucha gente confunde el constructor y cree erróneamente que es el encargado de construir el objeto. Esto es falso. El constructor no se encarga de instanciar el objeto, sólo se invoca en forma automática en el momento en que el objeto se instancia. No debemos tener esta confusión.

Como podemos observar dentro del ejemplo, el constructor tiene acceso de tipo público. Esto es importante, ya que como se invoca automáticamente, el exterior necesitará tener acceso a él.
El constructor se encargará de solicitarle al usuario los datos necesarios. Cuando llevemos a cabo la instanciación, será de la siguiente forma:

```
prisma miPrisma = new prisma();
```

Si compilamos el programa, obtendremos el siguiente resultado:

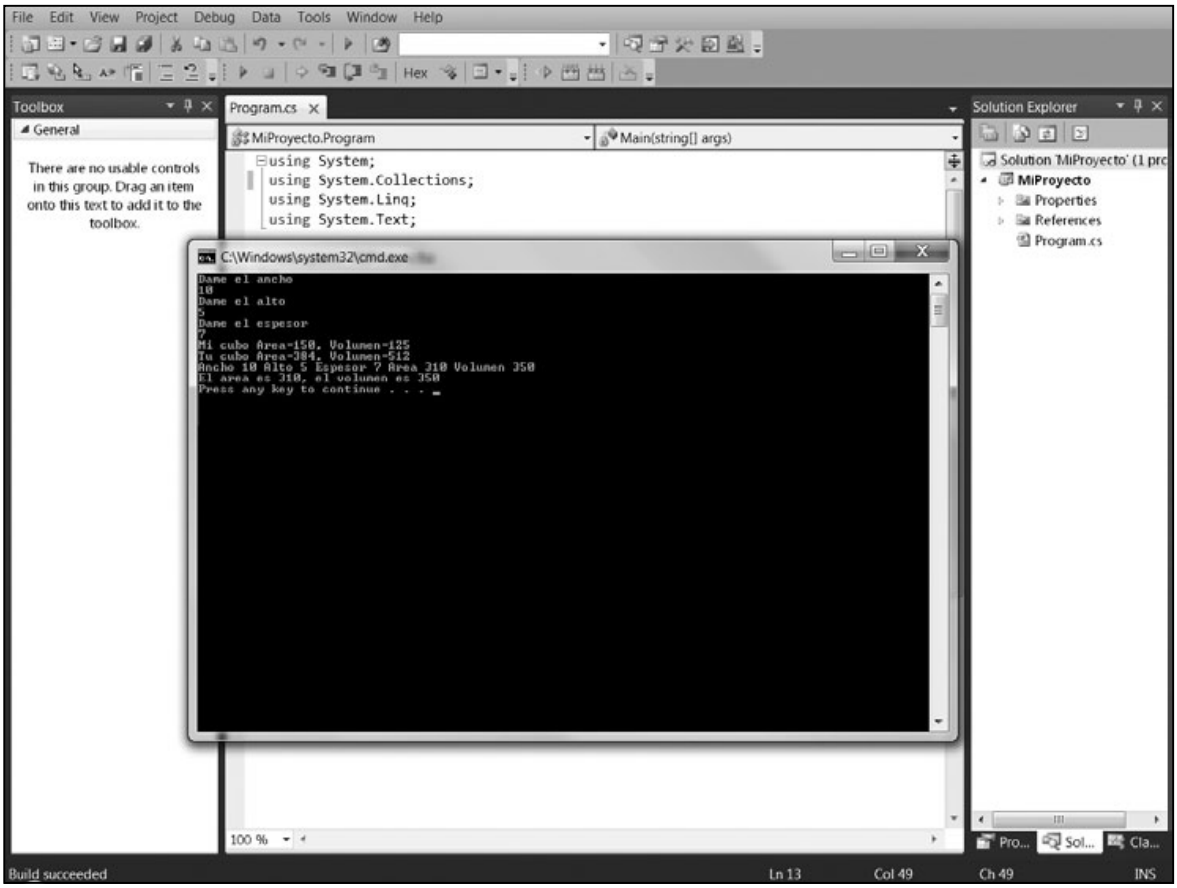

Figura 7. *Podemos observar cómo se ejecuta el constructor y nos solicita los datos necesarios.*

En las técnicas avanzadas de programación, como por ejemplo la programación de patrones, podemos encontrar constructores privados. Uno de los patrones que lo utilizan se conoce como **singleton**. Generalmente, haremos uso de constructores públicos hasta que aprendamos este tipo de técnicas avanzadas.

Sobrecarga del constructor

El constructor puede ser sobrecargado, es decir, podemos tener más de una versión del constructor. Esto resulta útil ya que podemos seleccionar cómo se inicializarán los datos del objeto dependiendo del tipo de constructor que utilicemos.
El compilador seleccionará automáticamente el tipo de constructor dependiendo de los tipos y la cantidad de parámetros.
Ya tenemos un constructor que nos pide directamente los datos, pero ahora podemos hacer un constructor que recibe los valores con los que se inicializará el prisma en el momento de la instanciación. Para esto tenemos que colocar parámetros en la segunda versión del constructor.

```
public prisma(int pancho, int palto, int pespesor)
{

     // Asignamos los valores
     ancho=pancho;
     alto=palto;
     espesor=pespesor;
}
```

Cuando instanciemos el objeto con este constructor, lo usaremos así:

```
prisma miPrisma3 = new prisma(5,3,7);
```

Observemos cómo los valores fueron pasados al momento de la instanciación. El valor **5** será colocado en el dato **ancho**, **3** en el dato **alto** y **7** en el dato **espesor.**
Si tenemos el dato contenido en una variable, también es posible utilizarla cuando instanciamos el objeto. Solamente debemos asegurarnos de que el tipo de la variable sea el mismo que el tipo del parámetro que colocamos. Si no fueran del mismo tipo, lo recomendable es utilizar **type cast**. Supongamos que tenemos lo siguiente:

```
int miNumero=11;
…
…
prisma miPrisma4 = new prisma(5,3,miNumero);
```

Como vemos, pasamos una copia del valor de **miNumero** y ahora en el interior del objeto **miPrimsa4** en dato espesor tendrá el valor de **11.**

Podemos continuar llevando a cabo más sobrecargas del constructor, tantas como sean necesarias. La cantidad de sobrecargas dependerá del análisis y las necesidades del programa. No debemos exagerar en las sobrecargas, debemos colocar solamente aquellas que sean realmente necesarias.

El programa completo que tenemos luce de la siguiente manera:

```
class cubo
{
    // Declaramos los datos
    public int lado;
    public int area;
    public int volumen;

    // Método para calcular el área
    public void CalculaArea()
    {
        area = (lado * lado) * 6;
    }

    // Método para calcular el volumen
    public void CalculaVolumen()
    {
        volumen = lado * lado * lado;
    }
}

class prisma
{
    // Declaramos los datos
    private int ancho;
    private int alto;
    private int espesor;
    private int area;
    private int volumen;

    // Definimos las propiedades
    public int Ancho
    {
        get
```

```
        {
            return ancho;
        }

        set
        {
            if (value <= 0)
                ancho = 1;
            else
                ancho = value;

        }

    }

    public int Alto
    {
        get
        {
            return alto;
        }

        set
        {
            if (value <= 0)
                alto = 1;
            else
                alto = value;

        }

    }

    public int Espesor
    {
        get
        {
            return espesor;
        }
```

```
            set
            {
                if (value <= 0)
                    espesor = 1;
                else
                    espesor = value;

            }

        }

        public int Area
        {
            get
            {
                return area;
            }
        }

        public int Volumen
        {
            get
            {
                return volumen;
            }
        }

        // Definimos los constructores

        public prisma()
        {
```

Las propiedades pueden tener cualquier nombre válido de C#. El nombre de las propiedades debe reflejar de alguna forma el tipo de dato sobre el cual actúa. Si nuestro dato es **costo,** entonces la propiedad se puede llamar **Costo**. El dato y la propiedad no pueden tener el mismo nombre, pero recordemos que C# distingue entre mayúsculas y minúsculas.

```
        // Datos necesarios
        String valor = "";

        // Pedimos los datos
        Console.WriteLine("Dame el ancho");
        valor = Console.ReadLine();
        ancho = Convert.ToInt32(valor);

        Console.WriteLine("Dame el alto");
        valor = Console.ReadLine();
        alto = Convert.ToInt32(valor);

        Console.WriteLine("Dame el espesor");
        valor = Console.ReadLine();
        espesor = Convert.ToInt32(valor);

    }

    // Version sobrecargada
    public prisma(int pancho, int palto, int pespesor)
    {

        // Asignamos los valores
        ancho = pancho;
        alto = palto;
        espesor = pespesor;
    }


    // Definimos los métodos
```

Tenemos que decidir oportunamente el tipo de acceso que permitirá la propiedad. Si tratamos de asignarle un valor a una propiedad de sólo lectura, el compilador nos indicará un error. Lo mismo sucede si tratamos de leer una propiedad de sólo escritura. Por eso lo mejor es planificar durante la etapa de análisis el acceso que les colocaremos a las propiedades de nuestra clase.

```
        public void CalculaVolumen()
        {
            volumen = ancho * alto * espesor;
        }

        public void CalculaArea()
        {
            int a1 = 0, a2 = 0, a3 = 0;

            a1 = 2 * CalculaRectangulo(ancho, alto);
            a2 = 2 * CalculaRectangulo(ancho, espesor);
            a3 = 2 * CalculaRectangulo(alto, espesor);

            area = a1 + a2 + a3;
        }

        private int CalculaRectangulo(int a, int b)
        {
            return (a * b);
        }

        public override string ToString()
        {
            String mensaje = "";
            mensaje += "Ancho " + ancho.ToString() + " Alto " +
               alto.ToString() + " Espesor " + espesor.ToString();
            mensaje += " Area " + area.ToString() + " Volumen " +
               volumen.ToString();
            return mensaje;
        }

        public void ImprimeResultado()
        {
           Console.WriteLine("El área es {0}, el volumen es {1}", area,
              volumen);
        }

    }

    class Program
```

```
{
    static void Main(string[] args)
    {
        // Instanciamos a la clase cubo
        cubo miCubo = new cubo();
        cubo tuCubo = new cubo();

        // Instanciamos el prisma
        prisma miPrisma = new prisma();

        // Instanciamos con la versión sobrecargada
        prisma miPrisma2 = new prisma(3, 5, 7);

        // Asignamos el valor del lado
        miCubo.lado = 5;
        tuCubo.lado = 8;

        // Invocamos los métodos
        miCubo.CalculaArea();
        miCubo.CalculaVolumen();
        tuCubo.CalculaArea();
        tuCubo.CalculaVolumen();

        // Asignamos los valores al prisma
        // Quitar comentarios para versión sin constructor
        //miPrisma.Ancho = 10;
        //miPrisma.Alto = 3;
        //miPrisma.Espesor = 5;

        // Invocamos los métodos del prisma
        miPrisma.CalculaArea();
```

Como siempre, podemos consultar MSDN para conocer más sobre los elementos que aprendemos en este libro. Si deseamos conocer más sobre las clases en C# podemos consultar el sitio web de Microsoft para interiorizarnos o profundizar conocimientos en el tema: **http://msdn2.microsoft.com/es-es/library/0b0thckt(VS.80).aspx**.

```
            miPrisma.CalculaVolumen();

            miPrisma2.CalculaArea();
            miPrisma2.CalculaVolumen();

            // Desplegamos los datos
            Console.WriteLine("Mi cubo Area={0}, Volumen={1}",
               miCubo.area, miCubo.volumen);
            Console.WriteLine("Tu cubo Area={0}, Volumen={1}",
               tuCubo.area, tuCubo.volumen);
            Console.WriteLine(miPrisma.ToString());
            miPrisma.ImprimeResultado();

            Console.WriteLine(miPrisma2.ToString());
        }
    }
```

Hemos escrito bastante código para el funcionamiento de esta aplicación. Ejecutemos el programa y nuestro resultado es el siguiente:

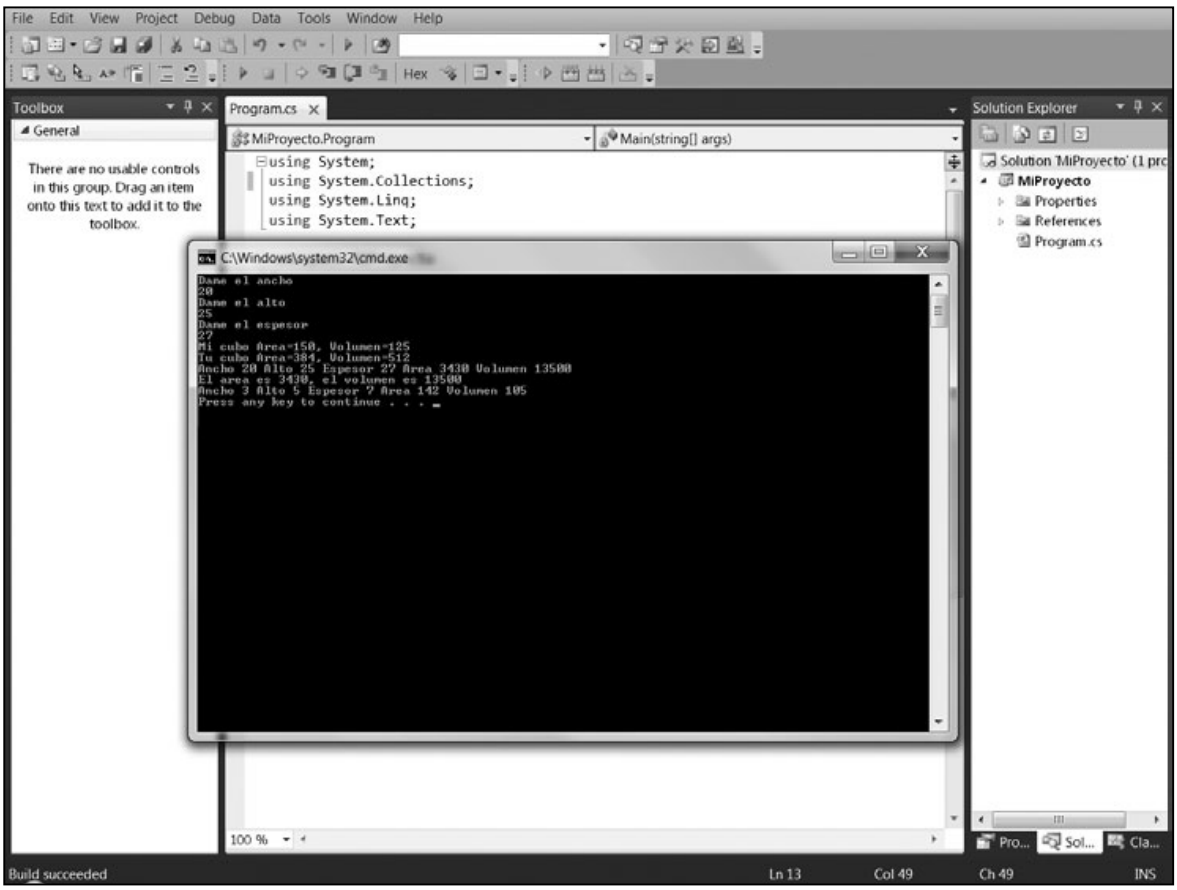

Figura 8. *Cada objeto usa el constructor que le corresponde.*

Si revisamos el código que hemos presentado en el bloque anterior, podemos darnos cuenta del funcionamiento de las sobrecargas del constructor que hemos agregado, es importante reafirmar la idea que ya explicábamos en en párrafos anteriores, tenemos la posibilidad de agregar cuantas sobrecargas sean necesarias pero siemre debemos tener en cuenta la necesidad de no exagerar.

A través de este capítulo hemos analizado la importancia del uso de clases para la generación de nuestros programas, utilizando C#. Con esto hemos visto el inicio de los elementos para programar clases. Aún queda mucho por aprender sobre la programación orientada a objetos y su programación en C#, pero éstos serán temas de un libro más avanzado y especializado en la programación del experto en el lenguaje C#.

RESUMEN

La programación orientada a objetos es un paradigma de programación diferente a la programación estructurada. Tenemos que reconocer los objetos que componen el sistema así como la comunicación que tienen entre ellos. La clase es el elemento fundamental de este tipo de programación y actúa como el plano sobre el que los objetos son construidos. Los objetos o las instancias son los que realmente llevan a cabo el trabajo. Los datos y los métodos pueden tener diferentes tipos de acceso: público, privado o protegido. Las propiedades nos sirven como funciones de interfaz para poder acceder a los datos privados de forma segura. Las clases pueden tener un constructor que nos ayuda a inicializar la información. Es posible hacer uso de la sobrecarga en el constructor.